In: Proceedings of the IFIP WG2.1 Working Conference on Program Specification and Transformation, Bad Toelz, Germany, April 1986, pp. 165-195.

A Survey and Classification of some Program Transformation Approaches and Techniques

Martin S. Feather
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey CA 90292
USA*

Abstract

Program transformation is a means to formally develop efficient programs from lucid specifications. A representative sample of the diverse range of program transformation research is classified into several different approaches based upon the motivations for and styles of constructing such formal developments. Individual techniques for supporting construction of developments are also surveyed, and are related to the various approaches.

* The author has been supported in part by the National Science Foundation under contract MCS-7918792 and in part by Defense Advanced Research Projects Agency contract MDA903 81 C 0335. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of NSF, DARPA, the U.S. Government, or any other person or agency connected with them.

* The author has been supported in part by the National Science Foundation under contract MCS-7918792 and in part by Defense Advanced Research Projects Agency contract MDA903 81 C 0335. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of NSF, DARPA, the U.S. Government, or any other person or agency connected with them.

1. Introduction

Program transformation has been advocated as the linchpin of an alternative programming paradigm in which the development from specification to implementation is a formal, mechanically supported process. The long range objective of this paradigm is to dramatically improve the construction, reliability, maintenance, and extensibility of software. Eloquent arguments along these lines may be found in [Balzer, Goldman & Wile 76], [Balzer, Cheatham & Green 83], [Bauer 76], and [Scherlis & Scott 83].

The current state of the art of program transformation is still some distance from supporting these ambitious goals, and research continues along a variety of diverse paths. The aim of this survey is to classify past and present trends in transformation research. Classification is done for **approaches** to program transformation, section 2, and for **techniques** for supporting program transformation, section 3. [Burstall & Feather 77] provides a much earlier survey of transformation research.

2. Approaches

The common thread of all transformation research is the **formal development** from specification to implementation. Because it is **formal** it offers the potential for introducing extensive mechanized support into much of the programming process. Because it is a **development** it records the complete path from the descriptive nature of the specification (which outlines what is to be achieved), to the prescriptive nature of the program (which details how things are actually done).

Two distinct goals are evident in most of the transformation efforts to date: the goal of **construction**, in which the aim is to improve the process of programming, and the goal of **comprehension**, in which the aim is to improve understanding of programs and algorithms. While these goals are related, the bulk of transformation research can be divided on the basis of which of these goals is the primary objective. Manifestations of this division are apparent in the degree of mechanization that has been sought, and in the nature of the example developments that have been performed. The construction goal has inspired the building of transformation systems to experiment with the mechanically assisted development of a broad range of programs. In contrast, the comprehension goal has inspired more hand-conducted investigations into explaining and understanding complex algorithms, and has had a lesser emphasis on realizing the techniques in any mechanical form.

Work that strives toward the comprehension goal is described in section 2.4. The other category (whose goal is construction), is further subdivided, forming its subcategories by comparing the varieties of transformational approaches with traditional compilation. Just as compilation develops efficient code from source programs, transformation develops efficient programs from specifications. Traditional compilation is characterized by the following key restrictions:

- Compilation is **automatic**, that is, once begun the process continues without any additional input from the programmer, and the product of the process is ready for execution.
- The **source language is limited** to be that which a compiler can "cover", that is, any program that is legal to express in the language should be compiled satisfactorily.
- Compilation begins with the source program only, and needs no advice on how to proceed.

By relaxing combinations of these restrictions in certain ways we get the major subcategories of transformation research aiming towards the construction goal. The identifiable subcategories are:

- **Extended Compilation**, characterized by permitting advice and partially relaxing the limits on the source language. That is, the transformation system accepts not only the source program, but also guidance on how to do the transformation. The source language is extended, taking advantage of the additional guidance to permit use of more expressive

constructs. Generally, however, such extensions remain fairly conservative, in that although there is a willingness to live with the possibility of some esoteric combinations of language constructs that cannot be satisfactorily compiled, most expected uses of this extra expressive power should be handled by the compiler. Note that by retaining the restriction that compilation be automatic, the results of transformation are ready for execution (in particular, will not need to be further transformed), and all guidance must be given at the very beginning along with the program. This category is explored in section 2.1.

- **Metaprogramming**, characterized by relaxing the restrictions that the process be automatic and that it take no advice. Now the transformation process may involve significant interaction between system and programmer, meaning that advice can be provided both at the start of, and during, transformation. As a result, coverage of the specification language can be attained without significantly limiting the constructs of that language. Transformation is best viewed as a manipulation of objects which are specifications or programs (there is no clear distinction between the two). Transformation systems execute the manipulations, guided by the advice from the programmer. Such guidance may be expressed as a structured series of commands to the transformation system, that is, advice takes the form of a program to cause the transformation system to manipulate its data objects, which are themselves specifications and programs. Hence the **metaprogramming** viewpoint. Drawing this analogy with programming has been a motivation for advances within this category of transformation research, considered in section 2.2.
- **Synthesis**, characterized by relaxing all language restrictions while attempting to retain the automatic and unadvised nature of compilation. To achieve this, the transformation system represents, manipulates and automates as much as possible of the knowledge that goes into constructing the development. Artificial Intelligence techniques are often applied to do this. In contrast to traditional compilation, the source language is left completely unrestricted, and the resulting language coverage that these mechanisms actually achieve is quite sparse. A specification may be expressed in a style that gives no hint of a reasonable efficient implementation. Thus the need to **synthesize** the program. Because the reasoning behind synthesis is captured within the system, advice from the programmer may be directions to the reasoning process, in addition to more specific directions of which transformation to apply next. Section 2.3 covers this approach to transformation.

2.1 Extended Compilation

This approach is characterized by being like traditional compilation in requiring that the transformation process be fully automatic, that is, once begun requires no further input from the programmer, and results in a program suited for execution without further examination or transformation, but differs from compilation by accepting advice on how to do the transformation (e.g., choices of data representations), and relaxing slightly the expectation that all legal specifications can be successfully transformed into efficient programs.

Since all advice must be given at the start of the transformation process, the programmer providing that advice must be able to express all of it at once, and anticipate in advance the interdependencies among those pieces of advice. This limits the degree to which specification languages may be extended beyond programming languages and yet still be amenable to this kind of transformation.

Research that fits in this category tends to be close to the boundary of current compiler capabilities. High-level languages are conservatively extended with a few additional constructs, some but not all uses of which can be transformed automatically given some initial advice. Generally, extended compilation is not able to radically restructure the algorithmic nature of the

specification. Hence writing specifications is very much like programming, except the specification languages are somewhat more expressive than programming languages.

By following this conservative approach to transformation, such research often has the considerable benefit of being immediately applicable. The researchers can and do use their systems to support their day-to-day programming. In contrast, the other, more ambitious, transformation approaches tend to be some considerable distance from the point of cost effectiveness.

Specifications in the extended compilers' carefully enhanced specification languages typically offer a wide range of choices of implementations, even while preserving the overall algorithmic nature implicit in the given specification. Conventional compilation tends to deal with simpler and/or better understood choices. For example, dead-code removal is clearly beneficial, so a compiler need only be concerned with doing it, not whether to do it; register allocation is done independently of (after) other optimizations, so the compiler need focus only on this one aspect at one time. In contrast, extended compilers may need to make explicit the space of all optimization choices so as to be able to evaluate and compare the efficiency of alternatives.

2.1.1 SETL

This long-running project at the Courant Institute of New York University [Dewar et al 81] has served as the context for a wide variety of transformation research. Their **very high-level programming languag**e, SETL, has syntax and semantics based on standard set-theoretic dictions of mathematics. SETL programs can always be executed; however, naive execution of programs that make liberal use of the high-level language features may be very inefficient. The SETL compiler has been built to compile SETL programs into efficient interpretable code or machine code. Used in this manner, the SETL compiler would fall into the category of a traditional compiler, albeit a very sophisticated one. Pertinent to the theme of extended compilation is the optimizer (described in [Freudenberger et al 83]) that lies at the heart of the SETL compiler. The optimizer improves SETL programs in the following ways: (1) run-time type checking is removed as much as possible by compile-time determination of variable types (SETL has dynamic typing, and makes heav:y use of overloaded operators); (2) efficient data-representations are chosen for SETL structures; (3) copying of objects is eliminated when it can be determined to be safe to transfer pointers rather than copies of values (SETL has sharing and side effects); and (4) classical “low-level” optimizations (common-sub expression elimination, etc) are performed. All the above can be done entirely automatically, i.e., compiled in the conventional sense.

This research crosses over into the world of extended compilation by allowing the programmer to provide declarations to direct the choice or representations ror data structures. These declarations are made in a “representation sublanguage.” which can be used to express a wide range of data structuring techniques. These do not cause the optimizer to make any significant change to the algorithmic form of the program; they merely direct its selection of data representations to support the provided algorithm. The representation sublanguage and its use is described in [Dewar et al 7g], along with some ideas on how to use global analysis[1] to automate the selection of efficient data structures. Research into more radical algorithmic changes has also been done in the SETL framework, and will be mentioned later.

2.1.2 RAPTS

Paige's RAPTS (Rutgers Abstract Program Transformation System) is a running transformational programming system that does source-to-source transformations on high-level SETL programs (Section 2.1.1). His overall approach and views are outlined in [Paige 83] and [Paige 84].

[1] Analysis is also used in [Wegbreit 76] to single out those portions of the specification appropriate for optimization.

RAPTS places **severe restrictions on the SETL specifications** that it can deal with, and relies upon a small amount of advice from the programmer to guide the transformation process. As a result, RAPTS can transform **very high-level specifications into efficient code**, introducing and radically altering algorithmic structures as it does so. The transformation process is divided into the following main stages: (1) **introduction of computability**; the original specification may be expressed in terms of infinite computations and infinite data objects --this stage transforms the specification into terminating computations on finite data representations, (2) **introduction of search strategy**; applicative expressions of searches are transformed into imperative algorithms committed to particular algorithmic search strategies (e.g., depth-first search); and (3) **improvements to the chosen strategy**; these are achieved by dynamic expression formation, formal differentiation (Section 3.2.3.2.3), loop fusion (Section 3.2.3.1.1), and dead code elimination. Once at this level, the remaining transformations are in the domain of the SETL optimizer. Paige remarks that having the record of how this level was achieved (through transformation from much higher-level stages) is of considerable utility, since it relieves the optimizer of rediscovering much of the information through intricate control flow analysis, and since the preceding transformations have ensured the absence of certain inefficiencies that would otherwise have to be considered (e.g., unnecessary copying).

RAPTS **derives a complexity formula for the specifications** that it transforms. This capability is uncommon in transformation research, and Paige argues strongly for its utility. Also uncommon is the very high-level nature of the specifications that RAPTS can deal with while still remaining a highly automated system.

2.1.3 TAMPR

Boyle's TAMPR (Transformation-Assisted Multiple Program Realization) system provides a variety of support for programming in FORTRAN at the Argonne National Laboratory, [Boyle 79] and [Boyle 84]. Applications include small **language extensions** (e.g., complex and quaternion abstract data types, automatic declaration of undeclared variables), **optimizations** (e.g., loop unrolling, unfolding of some subroutine calls), **conversions** (e.g., single to double precision, multi-dimensional arrays to one-dimensional ones), and **miscellaneous support** (e.g., instrumenting programs, recognizing inherent program structure). The modest nature of the tasks enables TAMPR ‘stransformation process to be entirely automatic. In addition to transformation within the FORTRAN language, TAMPR has also been applied to help in FORTRAN-to-Pascal translation, and in converting the bulk of the TAMPR system itself from its (almost) pure applicative LISP version into FORTRAN (which runs faster than the compiled LISP form on the same machine). This latter application demonstrates the feasibility of the approach on moderately large programs (1300 lines, 42 functions, converted into 3000 lines of FORTRAN). Boyle stresses that approaching these tasks by means of program transformation encourages organizing it in a modular fashion, with many consequent benefits. This issue will be discussed later, in Section 3.2.4.1.

2.2 Metaprogramming

In this approach, the transformation of a specification into a program is done interactively, by having the transformation system be guided by the programmer. As a result, ambitious specifications (that make liberal and widespread use of very high-level constructs, often requiring radical algorithmic changes to obtain tolerable efficiency) can be tackled --much more ambitious than is possible in the extended compilation approach. This raises the level for expressing specifications well beyond that of high level programming languages. However, the guidance that the programmer must provide to the transformation process becomes the critical aspect of this approach. For all but

the most trivial of tasks, the guidance itself is relatively intricate, and has to be constructed and refined through experimenting with its effect on the specification.

The record of the guidance provided by the programmer is now more complex than merely advice expressed in terms of the initial specification, as was the case in the extended compilation approach. Research following this approach has tended to draw an analogy between the transformation process and programming: the guidance that directs transformational development is like a program, executed on a transformation machine, whose data is the specification being transformed. In transforming specifications that are larger or more complex than the simplest "toy" problems, the transformational developments may be quite lengthy. This is because the transformation steps tend to be low level, in that they make only small manipulations to the specification and so a long sequence of such steps is required to build the entire development. This not only renders construction of the development quite tedious, but also means that the linear record of transformation applications is difficult to comprehend and lacking in robustness when replayed for maintenance purposes. To overcome these problems, researchers have sought to: (1) **structure** the transformational developments (in this case, the metaprograms), and (2) **mechanize** the lower level aspects of applying transformations. The desire to impose structure cuts across the transformational approaches; we will also see its prominence in developments for comprehension (Section 2.4). Similarly, the adoption of mechanization to assist in the transformation process is a common desire (the extended compilation theme is clearly committed to this; much of the synthesis research has been directed to realizing automated systems). The goal is to achieve a symbiosis between the talents of a skilled human, who is better able to make the strategic development choices, and the mechanical abilities of the system to flawlessly carry out the numerous trivial low-level manipulations. In the systems constructed to date, the level of diction between human and system has remained quite low. As a result, transformation done in this manner is still a tedious process, and the resulting development lacks the robustness necessary for successful application of the transformational paradigm for maintenance. Those researchers who have done metaprogramming-style developments by hand have had the advantage of more flexibility, most notably in their introduction and use of sophisticated notations, but suffer the disadvantage of lacking mechanical support.

Having obtained a program that records how to transform the specification into an efficient program, such a (formal) record may serve the following purposes: (1) understanding the efficient program may be achieved by beginning with the (presumably lucid) specification and then studying the development, which gives an evolutionary view of how efficiency is introduced, and (2) maintaining the efficient program --when the program is to be changed, it is done by modifying the development (to adjust the efficiency characteristics without altering the functional behavior), or by modifying the specification and reperforming the development. In the latter case, the hope is that much of the previous development will be reusable. Again, the formal record of the development and the reliance upon programmer assistance enhance the feasibility of maintenance performed in this manner.

2.2.1 ZAP

Feather's ZAP system and language [Feather 82a] is based on the fold/unfold work of Burstall and Darlington (Section 3.2.1) on transforming applicative programs expressed in recursion equations. The ZAP system IS language is an (ad-hoc and relatively primitive) **language for expressing transformational developments**. ZAP is applied to several modest problems (the telegram problem, portions of a toy compiler, and a small text formatter [Feather 82b]). Each of these began with a computationally naive specification of significant size, and so the transformation process exercised the need to deal with issues of scale because the developments were quite lengthy.

Some **higher level means of structuring the developments** are applied informally, but not represented within the ZAP language. The calling structure of the specification's functions is used to suggest an overall strategy for efficiency improvement, where the elements of the strategy are applications of fusion, tupling, and generalization tactics (Section 3.2.3). The tactics are in turn expanded into sequences of transformations at the fold/unfold level. Expansion from strategy to tactics is done entirely by hand, and is not represented within the ZAP language nor supported by the system. Automated mechanisms provide some support for expanding tactics into the fold/unfold sequences. The most useful of these provides "pattern-directed transformation," a means of describing the approximate form of the next stage of development, leaving the mechanism to discover the sequence of transformations that will effect the appropriate change. In addition, there are heuristics that suggest plausible case breakdowns and recursions.

The issues in **applying program transformation to the maintenance of software** is discussed in [Darlington & Feather 80], where the ZAP system is applied to retransform some modified small scale specifications.

2.2.2 HOPE metalanguage

Darlington pursues the idea of a language for recording developments and draws inspiration from the LCF project[2] [Darlington 81a]. The applicative programming language HOPE is adopted as the **metalanguage in which to express developments**. Starting with a small set of basic transformation rules, HOPE's structure-defining and manipulating facilities are used to express higher level transformation rules in terms of lower level ones. Hierarchically composed rules, transformation "tactics" (e.g., merging loops, recursion to iteration), and (potentially) algorithmic paradigms (e.g., binary search) are all expressed as HOPE operators. The development is then a HOPE program that applies the appropriate operators to effect the desired transformation. Darlington has applied this approach to some transformational developments of small examples.

The **structuring of design decisions** has also been investigated by Sintzoff [Sintzoff 80]. He makes suggestions for hierarchically structuring program designs, demonstrated on some hand-performed case-studies.

2.2.3 POPART/Paddle

Wile's POPART system [Wile 82] supports experiments with developments, transformations, and other program manipulations. Pertinent to the metaprogramming theme is POPART's special-purpose language Paddle, designed to be used to **express the structure of developments** [Wile 83]. As with the use of HOPE as a metalanguage, in Paddle one may compose transformations to build structured developments. Transformation rules are divided into four categories: (1) simplification rules (which are applied automatically after major changes), (2) conditioning rules (intended to be invoked automatically in preparation for major transformation steps, Section 3.2.1), (3) a catalogue of commonly useful rules (invoked by name), and (4) the Paddle program that is the development for the task.

In addition to expressing the subordinate structure, Paddle also provides for expressing the **goal dependency structure of the development.** (Examples of development goals themselves are: attain maximum efficiency, eliminate recursion, and merge adjacent loops iterating over the same structure.) Wile argues for the importance of recording such dependency, particularly to **facilitate reuse of the development**. For example, if two subgoals are recorded as being independent, and the

[2] Where a **metalanguage**, MI., was developed to allow the **writing or structured plans to assist in the proving or theorems about programs** [Gordon et al 78]. The PRL program development system also provides mechanical support for constructing proofs, proofs in a form that may serve as programs [Bates & Constable 85]

specification changes in such a way as to affect only one of them, it will be clear that the other subgoal can be reapplied without change. Paddle can record the following goal structures:

- sequential dependency (goal A must be achieved before goal B),
- independence (goal A may be achieved in parallel with goal B),
- choice (goal A was chosen from a set of possible goals {A,B,...}, all of which supported the same overall goal),
- conditionality (goal A need only be achieved if goal B could not be achieved), and
- repetition (apply goal A to all instances of...).

The Paddle/POPART combination has been applied extensively to construct and apply transformational developments to a number of moderate-sized specifications (e.g., a text-compression program and a package router, a small process control problem).

2.2.4 CIP

The long-running CIP project (Computer-aided, Intuition-guided Programming), at the Technical University of Munich, has contributed much to the field of transformation, and references to various facets of their work occur throughout this survey. The methodology that they espouse for program development fits into this metaprogramming category. See [CIP 84] for a recent summary of the project, and extensive references.

They have a running prototype of a transformation system, and are in the process of using it to help implement a new transformation system which will provide the mechanical support for user-directed transformational developments. The new system has been formally specified, and this is serving as the starting point for a transformational development of the implementation, [Partsch 84a] and [Horsch et al 85]. Their metalanguage for expressing developments, like HOPE and Paddle, permits the definition of more complex rules in terms of the elementary rules [Steinbruggen 82]. It is anticipated that the vocabulary of the metalanguage will be enriched as experience is gained from its use.

2.3 Synthesis

The objective of the synthesis approach is to make transformation as automatic as possible without limiting the specification language in any way. To achieve these seemingly contradictory objectives, synthesis research must **represent, manipulate, and automate as much as possible of the knowledge that goes into constructing a development**.

As with metaprogramming, specifications may be expressed in a style that gives no hint as to any reasonably efficient programs (indeed, it is common to start with specifications that are in a non-executable form). Synthesis must therefore tackle all the problems that are encountered in the metaprogramming theme, but instead of there being a skilled programmer exterior to the system to turn to for guidance, the transformation system itself must do deep reasoning about the development process. Such systems expend most of their effort in expanding and navigating through the space of possible development alternatives. Artificial Intelligence techniques (particularly those drawn from planning and theorem proving) are often applied to represent and organize such searches. The very ambitious nature of these goals has limited the success of automatic synthesis systems to very small tasks or within a very limited domain of tasks. Existing such systems should be seen as experimental vehicles for testing ideas, not as prototypes of future tools for support of software construction.

There is a place for interaction between a programmer and a synthesis-style transformation system, although now the advice is not expected to take the form of particular transformation applications (bare advice in this form would be hard to reconcile with the reasoning process being built up by the system) but rather would be of new implementation strategies, tradeoffs among

existing ones, etc. An ideal system along these lines would converse with the skilled programmer in high-level terms about development goals and objectives.

2.3.1 Manna & Waldinger

Manna and Waldinger were pioneers in this field, and have explored a variety of approaches to synthesis. Their efforts divide into those that **organize the construction or the development around the rorm or the specification as it is manipulated toward a .program, and those that are rounded upon theoremproving techniques**, in which the program emerges as a by-product of the theorem-proving effort.

Their efforts in the former direction, reported in [Manna & Waldinger 79] and [Waldinger 77], deal with small tasks expressed as relationships between inputs and outputs. Generally, synthesis begins with a mathematical description of the relationship, and may necessitate introducing conditionals, loops, recursion (including mutual recursion), and assignments to variables. The end-products are programs in a LISP-like language. In conjunction with producing the program, their techniques also construct proofs of termination where appropriate. Examples include simple numerical programs (algorithms for computing the greatest common divisor starting from a non-constructive definition), small searching and testing problems in the domains of lists and sets, and small "structure-changing" problems (e.g., finding the maximum element in an array, where changing the contents of the variable that is to hold the output is allowed, but changing the input array is forbidden). They have realized combinations of some of these abilities in automatic systems that naively explore the synthesis space in a backtracking manner.

Dershowitz [Dershowitz 85] continues in the same spirit to investigate the **formation or iterative loops**. His largest synthesis is of a partition problem (given an array and a position within that array, rearrange the elements so that each element at or to the left of that position is less than or equal to each element to the right of that position).

Manna and Waldinger's other efforts seek to apply the power and approaches of automatic theorem provers to the synthesis task. **The program is formed in parallel with building a constructive-style proof** that establishes the existence of an output satisfying the specification. The proof effort is used to direct the synthesis, and may employ traditional proof techniques (e.g., induction, generalization). Although their studies along these lines have not been embodied in an automatic system, they are close to the level of detail necessary for such mechanization. See [Manna & Waldinger 80] for details. Their most ambitious synthesis is of a unification algorithm, described in [Manna & Waldinger 81]. This synthesis is entirely hand performed, although it is done at a fine enough level of detail to consider the capabilities that would be required of a mechanical system to perform the development.

2.3.2 PSI (PECOS & LIBRA / CHI)

The PSI program synthesis system [Green 77] was constructed at Stanford to develop and test ideas for supporting symbolic programming. The PSI system supported the acquisition of specifications and their development into efficient programs. The latter phase requires **cooperation between two knowledge-based programs**: PECOS, which **generates implementations in a stepwise refinement fashion**, and LIBRA, which **efficiently directs PECOS through the search space of alternative refinements** toward an efficient implementation.

PECOS [Barstow 79] operates with a large **catalog of transformation rules** about symbolic programming. Most of the catalog comprises refinement rules, which generally refine a data structure or abstract operation into a more concrete implementation (e.g., a sequential collection may be refined into a linked list); the catalog also contains **property rules**, which are applied to annotate the developing program with additional information, and **query rules**, which are applied to answer

questions that arise during refinement (e.g., satisfaction of applicability conditions). These rules typically effect only very small changes, and hence a long sequence of rule applications may be required to develop a small specification into a program. The disadvantage of having to control search for these long sequences is offset by the advantages of enhancing the coverage and extensibility of the rule catalog. By isolating the individual programming decisions into separate rules, those rules can be combined in many different ways to yield a wide variety of implementations; an added rule may be combinable in many ways with existing rules. Barstow [Barstow 85] argues that experiments in the domains of elementary symbolic programming and graph representations suggest convergence toward a useful catalog of rules, that is, as successively harder tasks are transformed, fewer and fewer new rules need be added to the catalog to make possible the transformation of further tasks.

The PECOS style of synthesis has been hand applied to the development of a class of in-place sorting programs (bubble sort, sinking sort, quicksort, and mergesort) [Green & Barstow 78]. This is closely related to Darlington's development of the same programs (Section 2.4.1); however, here the emphasis is on explicating algorithm design principles and expressing them as refinement rules.

LIBRA, the other component of PSI's development phase, directs PECOS's expansion of and navigation through the search space of refinements, [Kant 83]. Since the space of alternative refinements is large (a development from specification to program may involve a long sequence of steps, and at each step along the way there may be a number of alternative refinements to consider), LIBRA is concerned with **conducting the search for an efficient implementation in an efficient manner**. LIBRA is provided with the initial specification, size and frequency notes about that specification (estimates of data structure sizes and probabilities of alternative branchings), a performance measure which it is to minimize (a polynomial in storage space and running time), and limits on resources that may be expended during synthesis (upper bounds on space and time). LIBRA follows a strategy of modified best-first search with lookahead, and employs a catalog of rules to embody search knowledge and analysis knowledge (for estimating the efficiency of partially refined programs). Some search knowledge rules are heuristics for improving the planning process independent of the refinement rules (e.g., to group related decisions together so as to reduce search and make cost tradeoffs more obvious), and some require knowledge of refinements (e.g., to identify plausible refinements out of a set of alternatives). Analysis knowledge rules are applied to estimate the cost of a partially refined program. Cost analysis is done incrementally in combination with the ongoing refinement process; two cost estimates are maintained, an upper bound (which is guaranteed to be achievable) and an “optimistic” lower bound, which represents the lower bound for implementations known to the coding rules, assuming no adverse interactions. (The PECOS/LIBRA combination does allow for synthesis to result in a program using multiple representations for the same data structure, and LIBRA takes into account the cost of representation changes.) LIBRA provides some semi-automatic facilities for extending its search and analysis rule base when PECOS' rule base of refinements is extended.

The PECOS/LIBRA combination has been applied to synthesize modest programs, including classification and retrieval programs, selecting data representations for a prime-number generating algorithm, and sorting. Two paradigms for combining PECOS and LIBRA have been tried. In the earlier paradigm, each time PECOS faced a choice, each refinement would be applied and the results passed to LIBRA for analysis and preferential ordering, after which PECOS would continue with the preferred alternative. Experiences with this paradigm are discussed in [Barstow & Kant 76]; briefly, the narrow channel of communication between the two components was found to lead to inefficiencies in the searching process. The second paradigm is the one described in the preceding paragraph: the tree of refinements under construction is shared between PECOS and LIBRA; LIBRA

essentially directs the exploration, using PECOS as a "legal move generator"; see [Kant & Barstow 81].

More recently, the approaches and techniques of the PSI system have been applied to construct a **knowledge-based programming environment**, called CHI, and to explore the creative aspects of algorithm design. Central to this environment is the wide-spectrum language V. For details, see [Green et al 81] and [Smith et al 85].

2.3.3 Glitter

Fickas's experimental Glitter system resulted from his investigations into automating much of the development process from high-level specifications (written in a general purpose specification language) to efficient programs, [Fickas 82]. His approach is to **make explicit, to formalize and to mechanize the planning aspects of development**.

The Glitter system comprises the following:

- a set of **goal descriptors**, used to express development goals (e.g., remove is a goal descriptor which, when associated with a construct and a context, describes the goal of removing all uses of the construct from the context);
- a catalog of **methods** that may be applied to achieve goals; these include conventional program transformations that manipulate the specification, and problem transformations that manipulate the planning process (e.g., refine an outstanding goal into a set of more manageable subgoals);
- a **method applier** that applies a method;
- a catalog of **selection rules**, used to direct the planning process by comparing competing methods in the context of the planning that has taken place so far; and
- a **problem solver** that finds methods to achieve goals, uses selection rules to order its search through those methods, and invokes the method applier to apply the methods it selects.

Ideally, the programmer's interaction with Glitter would be confined to providing development goals at timely intervals. In practice, the programmer is also required to provide more direction to the problem solver in the form of additional methods, justifications of applications of methods and selection rules, and explicit navigation through the development space.

Fickas has applied Glitter to a modest process control example (development of a controller for a postal package router).

2.3.4 Operationalization

Mostow has considered the synthesis task in the case of **specifications expressed in terms of data and activities not directly available to the implementation**. He calls the development of an effective program from such a specification "operationalization." This typically arises when the task is to implement a component that resides within, and interacts with, an environment; specification of that task may be in terms of data within the environment (which need not necessarily be directly accessible by the component) and may be of activities of the environment (which need not necessarily be under the direct control of the component).

His first system, FOO, interactively constructs operationalization-style developments. The development space is **formalized in traditional AI terms as a problem space**; FOO is equipped with a catalog of rules that encode operators to move within that space. The system interacts with a skilled programmer to direct the application of these rules. Using this, Mostow has operationalized individual heuristics of good play in the card game Hearts, and (for music) generation of a cantus firmus, a sequence of whole notes satisfying certain aesthetic constraints. These developments are detailed in [Mostow 81].

Mostow's follow-on system, BAR [Mostow 83], is designed to **automate most of the search through the development space of operationalization**. BAR makes the goal structure of operationalization developments explicit, and uses an AI-style means-ends analysis problem solver operating over this goal structure to automate the search. BAR has been applied to several of the developments done with FOO, and, although it still requires some interactive guidance from the programmer, it succeeds in vastly reducing the branching factor among alternatives within the development space.

There are several particularly interesting aspects to BAR's operation. It automatically analyzes the transformation rules to determine their effects with respect to the operationalization goal space (over which the planner searches). Also, program transformation rules are encoded independently of the formulation of the problem space. Together these imply that further rules (both program transformations, and more domain knowledge) may be added without any modification to the problem-solver. Since operationalization developments involve transformation sequences dozens of steps long, and with several rules applicable at each step, the problem solver's search space is combinatorially explosive. To mitigate this, search is done first in an (automatically) abstracted space of goals and transformations to yield successful abstract plans, which may then be refined into the more detailed developments.

2.4 Comprehension

The characterizing feature of this category is the goal of understanding complex programs and algorithms by developing them systematically from lucid specifications.

Generally, the kind of problem tackled in this manner is a concisely specifiable task for which there is an already known efficient algorithm of significant intricacy. The development is analogous to a constructive proof – not only does it convince the reader that the program does solve the task, it also demonstrates **how** it solves the task, by revealing the design decisions which lead to the final algorithmic form. Varying the design decisions during the course of the development may give rise to different algorithms to solve the same problem, thus exposing the relationships among these families of algorithms. These and other points are made in [Reif & Scherlis 83], in the discussion of the benefits of this type of research. Another motivation is to help in the verification of programs – it has been suggested that proofs of correctness be derived by proving the correctness of some high-level version of the program, and applying correctness-preserving transformations to develop the efficient low-level program, [Gerhart 75] and [Broy & Bruckner 80].

It is important to stress that the objective of this style of research is not the discovery of new algorithms[3] – usually the problems are well known, and have already received considerable attention (I.e., the known algorithm may have been analyzed for complexity and/or verified correct with respect to the specification).

Many of the problems that have been tackled in this fashion have been developments of quite intricate algorithms, and such research contributes to transformational expertise. Even though these developments are generally constructed by hand without any mechanized support, there are similarities between these efforts and those of the metaprogramming theme. Here, too, the transformational development is lengthy, to the degree where recording merely the linear sequence of transformations would be incomprehensible. Thus structuring becomes a necessity, and developments employ a rich set of dictions with which to express such structure.

Since there have been many developments in this style, space allows only a quick survey of a few of the more landmark ones that exemplify a variety of aspects.

[3] Though this has occurred; see Sharir's strange sorting algorithm [Sharir 81].

2.4.1 2.4.1. Families or sorting algorithms

An early result in this vein is Darlington's development or a family or sorting algorithms for lists and arrays (quicksort, merge, insertion, selection, exchange and bubble) from a single naive specification of sorting [Darlington 78]. The starting specification is in generate-and-test form, that is sorting is defined as the selection of an ordered member from all permutations of the input (a list of comparable objects). The overall structure of the development is to improve the efficiency of the permutation generator, and then to promote the selection (of an ordered member) into the generation. Varying the design decisions at this level gives rise to the family of sorting algorithms, and thus exposes their interrelationships. At the next level of detail, the transformations are organized around applications of key lemmas, generalizations, and goals of achieving certain recursive forms. These are revealed as the insightful points within these developments. At the lowest level of detail, the manipulations are done in the fold/unfold style of transformation (Section 3.2.1), which (apart from some use of notational extensions) is essentially at the level of detail of a transformation system. Development of families of sorting algorithms are also to be found in [Green & Barstow 78], and [Broy 83]. Other families of algorithms developed transformationally include transitive closure – [Schmitz 82], parsing and recognition – [Partsch 83a] and [Scherlis 80], and graph algorithms – section 2.4.4.

2.4.2 2.4.2. Schorr-Waite and Earley's recognizer -CIP

In the context of the CIP project (Section 2.2.4), the researchers have produced numerous and transformation-based developments of a variety of algorithms. Only the two most sophisticated developments that they have published to date are mentioned here.

The first is Broy and Pepper's development of the Schorr-Waite graph-marking algorithm [Broy & Pepper 82]. The starting point is a mathematical specification of the general problem (computing reflexive, transitive closure of a relation), the end point an efficient procedural program. The most notable aspects of this development are as follows:

- The development is divided into **major phases, based upon the various language levels** through which the expression of an emerging algorithm progresses. These levels are initial mathematical specification -> depth-first-search recursive algorithm -> functional (applicative) version of the Schorr-Waite algorithm -> nonrecursive procedural program.
- **Abstract data types** are used to express selective updating at the applicative level; this greatly facilitates reasoning and transformation of the graph pointer manipulations that lie at the heart of the algorithm.
- A **specially constructed and proven transformation rule** is applied to convert the recursive procedure into an iterative program. This rule can be used to generate a variety of versions of the algorithm, and its application is one of the key steps in the development.
- The idiom of **embedding** (a means of solving a special problem by considering a more general one; Section 3.2.3.1.3) is used repeatedly through the course of the development.

The second development, by Partsch, is of Earley's context-free recognition algorithm [Partsch 84b]. The major phases of the development are **conversions between forms of the specification**, in a manner similar to Broy and Pepper's development. Partsch's presentation **emphasises structuring**. He gives an overview of the major steps of the development, identifying for each the goal and the individual transformations applied to achieve that goal. Informal commentary accompanies each step. Having established the main structure, he provides more detail of the individual transformation steps (at the level of detail of a machine-checkable proof). Partsch also provides a comparison with development of the same algorithm, including one by Scherlis [Scherlis80],who shares the same overall methodology.

2.4.3 Evaluating linear recurrence relations - Pettorossi and Burstall

Pettorossi and Burstall [Pettorossi & Burstall 82] develop logarithmic-time algorithms for evaluation of homogeneous linear recurrence relations with constant coefficients. They do this by starting with a simpler, but analogous example, the Fibonnaci function. They first develop two different logarithmic-running-time algorithms for Fibonnaci. Each development is structured by breaking it into stages on the basis of the idiomatic effect of that stage of transformation (generalization, application of lemmas, tupling [Section 3.2.3.1.2], and simplification). Each stage is achieved by application of a sequence of fold/unfold manipulations. Development of an efficient program for the original problem, solving homogeneous linear recurrence relations, is then constructed by **drawing analogies** with the form of the Fibonnaci problem, and the idiomatic effects of the stages in its two developments.

2.4.4 Graph algorithms - Reif and Scherlis

Reif and Scherlis [Reif & Scherlis 83] develop Hopcroft and Tarjan's depth-first-search algorithms for computing biconnectivity and strong connectivity in graphs. They do this by first **developing a family of simple depth-first search algorithms**. These developments, and the algorithms that result, are then **used directly or by analogy** in the later developments.

The transformation steps are often combined to achieve Scherlis' "specialization" tactic (Section 3.2.3.1.3); use is also made of a "finite closure" transformation and many modest lemmas expressing domain-specific knowledge. Sundry low-level simplifications and manipulations are interspersed among these steps.

Scherlis has a view of the development process as **operating within a space of alternative evaluations of expressions**. The task is to evaluate some expression (parameterized by its inputs). Typically, there will be many alternative ways to perform such an evaluation; the initial specification usually denotes a straightforward but inefficient one. Viewed in this manner, development becomes the discovery and selection of equivalent but more efficient evaluations.

2.4.5 Garbage collection and compaction - SETL

Dewar, Sharir, and Weixelbaum [Dewar, Sharir & Weixelbaum 82] develop a variant of a known efficient garbage collection and compaction algorithm from an initial highlevel specification. The initial specification is a simple algorithm constructed to solve the problem without regard to efficiency. It and all successive versions are expressed in SETL, and as such are executable. The authors claim this to be an especially important property while the transformational process remains far away from full mechanization.

Their development is organized into a **short sequence of major steps, each of which prepares for and/or applies some sophisticated transformation**. The earlier transformations establish the overall algorithmic form of the final program. The later transformations optimize this algorithm in the manner of the SETL extended compiler (Section 2.1.1).

Within the SETL framework, the key transformations that introduce and modify the algorithmic form of the emerging program are: commitments to incremental computation, loop fusion (Section 3.2.3.1.1), splitting of a single computation into substeps, and formal differentiation (Section 3.2.3.2.3). Often the goal of applying one of these transformations motivates the preparatory application of one or more of the other transformations to get the program into a suitable form (Le., conditioning – Section 3.2.1). The authors make the claim that most of the transformations are routine, and hence that the development could be done largely automatically. One particular transformation step that splits apart a piece of the computation into two stages corresponding to "adjusting the pointers" and "moving the blocks" is singled out as perhaps requiring considerable insight. Its result embodies the essence of this family of garbage collection techniques, and although

the step is justified by technical transformation reasons, they suggest this to be fortuitous, and feel this to be the major insightful step of the development.

Stylistically, this development differs slightly from the preceding ones. Here the starting specification is not as concise, and has more of a procedural flavor. The concerns addressed during development also exhibit more of a procedural character. Nevertheless, the means of **structuring** the development and the **idioms** used in doing so have analogies across all the developments in this approach.

3. Techniques

In this section we consider some of the techniques that have been invented and applied to support the transformational development process. Support techniques are divided into the following categories:

- **Mechanical assistance** for effecting transformations is considered briefly in section 3.1.
- Lengthy developments are **structured**; a variety of structuring techniques are outlined in section 3.2.
- Special-purpose **languages** are used to express specifications and programs, transformational developments, and transformations themselves, section 3.3.

3.1 Mechanical support - transformation systems

Program transformation, by formalizing the development process, is intended to make software development amenable to mechanical support. The current state of program transformation research is such that mechanization is not, in most instances, cost effective as a means of software production. However, many researchers have built systems as experimental vehicles to test certain aspects of their transformation ideas. Brief references to various transformation systems are to be found throughout this paper; for a more complete and detailed survey, see [Partsch & Steinbruggen 83].

3.2 Structuring techniques

A common problem of most transformation efforts is the lengthy and low-level nature of developments. Structuring developments has long been recognized as essential for their construction, comprehension and maintenance, regardless of whether these activities are to be fully automatic (as in the extended compilation approach), semiautomatic (as in metaprogramming), or entirely manual (as in many of the comprehension goal developments). Identifiable structuring techniques that have been applied in developments include the following:

- Identifying certain transformation steps as **pivotal** can motivate the application of other transformations around those steps, section 3.2.1.
- The nature of the specification to be transformed, the specification language, and the target programming language, can be used to indicate an overall **strategy** for transformational development, section 3.2.2.
- The building blocks of a strategic development are transformation **tactics**, where a single tactic effects some particular kind of change to the specification undergoing transformation, section 3.2.3.
- Transformation **catalogs** can often be structured based upon the specification language, independent of the details of particular specifications, section 3.2.4.

3.2.1 Pivotal transformations

The fundamental idea of this structuring technique is to identify certain transformation steps as pivotal, and to organize the application of other transformations around such steps.

This phenomenon arises in the context of Burstall and Darlington's fold/unfold transformation method [Burstall & Darlington 77], used to transform applicative (side-effect-free) specifications expressed in a language of first-order recursion equations. Developments of even quite small specifications require the construction of very lengthy sequences of transformation applications. The transformations are selected from a small set of simple rules optionally extended with rewrite rules encoding domain knowledge. The most interesting simple rule is "fold", whose application replaces an instance of the body of a function with a call to that function, and in many developments the application of a fold rule is the **pivotal step**. Burstall and Darlington took advantage of this in their experimental transformation system, making it automatically apply other transformations (in a naive backtracking manner) until a fold step can be performed, after which the system pauses to query the user on whether or not to continue with that development path. Some of the transformation rules (the abstraction rule that makes a structural reorganization, and domain rules that express associativity and commutativity of operators) are **automatically applied only when needed** to make a fold possible. This enables the system to semi-automatically perform numerous small developments. Darlington extended the system with the capability to **introduce new functions** (whose definitions are more complex than trivial aggregations of existing functions) in order to **make a fold possible**, a technique he called "forced folding" [Darlington 75, Darlington 81b]. This extension, similar to controlled generalization (section 3.2.3.1.3), widens the range of programs that can be developed with the aid of this system.

More generally, any transformation application that is identified as pivotal may be used as a goal to **direct the search for, and the application of, other transformations**. Balzer, in describing the interactive construction of a transformational development [Balzer 81], suggests that the user-directed application of a transformation may serve as the goal whenever the transformation is inapplicable to the current state of the program, in which case the user is given the option of entering a subgoaling mode with the objective of applying other transformations to make possible the suspended transformation step. This idea, called "jittering" or "conditioning," was continued by Fickas in the framework of his goal-directed transformations (Section 2.3.3).

3.2.2 Transformation strategies

It is clear that the nature of the specification to be transformed, the specification language, the target programming language, and the efficiency requirements, may all influence the development from specification to program. Unfortunately, little is known about how to deal with all these issues at once in anything other than an ad-hoc manner.

Research that has addressed (but not solved) the problems at this strategic level includes:

- The structure of an applicative-Ianguage specification is used to suggest an overall strategy for efficiency improvement, where the elements of the strategy are applications of several transformation tactics, [Feather 82a].
- In an efficient implementation of a given specification, the optimal control structures and data structures may be mutually dependent. Guide-lines for intermixing the steps that focus on one or the other of control and data are outlined in [Partsch 83b].

It is often the case that an attempt at implementation (by transformation or otherwise) can lead to modification of the specification. This phenomenon is discussed in [Pepper & Partsch 80] and [Swartout & Balzer 82]. Again, almost nothing is known about how to deal with this, merely that it occurs.

3.2.3 Transformation tactics

The building blocks of a strategic development are transformation tactics, where a single tactic effects some particular kind of change to the specification undergoing transformation. The nature of

the change can be described abstractly, that is, independently of the particulars of the specification that is being transformed, and a description of the effect of a tactical change is often shorter that the sequence of corresponding transformations that actually execute that change. The following classes of tactics are considered:

- Tactics to introduce or alter the computation structure, section 3.2.3.1.
- Tactics to introduce or alter the maintenance and retrieval of data, section 3.2.3.2.
- Tactics to manipulate and implement abstract data types, section 3.2.3.2.4.

3.2.3.1 Transformation tactics on computation structure

This first class of tactics comprises those that affect the computation structure of the specification.

These changes predominate in the transformation of applicative programs and expressions, where most of the transformation sequences are intended to alter the calling structure among functions and establish an efficient computation order. Several idiomatic structural changes have been identified and applied in developments. Those receiving the most widespread use (across several of the transformation approaches) are: **fusion** (section 3.2.3.1.1), **tupling** (section 3.2.3.1.2), **generalization** (section 3.2.3.1.3), and **filter promotion** (section 3.2.3.1.4). **Removal** of certain forms of computation (notably recursion and nondeterminism) is another recurrent tactic (section 3.2.3.1.5, as is the **precomputation** of a program on some of its data (section 3.2.3.1.6).

3.2.3.1.1 Fusion

Fusion is the merging of nested function calls (in the context of recursion equation programs) or consecutive loops (in the context of iterative programs), where the first function call / loop builds up a composite object which is used by the second function call / loop. When this is the case, it may be possible to merge the two function calls / loops into one and thus avoid constructing the intermediate composite object. (In simple cases, much of the efficiency improvement can also be obtained by lazy evaluation.) The following are some examples of the widespread use of fusion (also called **composition** and **vertical jamming**) to structure developments:

- In the context of transforming applicative languages, fusion is one of the tactics mechanically supported by the ZAP system (section 2.2.1). In the same context, these are expressed as “second level transformation tactics” in the transformation metalanguage HOPE (Section 2.2.2).
- Descriptions of SETL transformational developments use loop fusion as an informal objective: [Sharir 82] and [Paige & Koenig 82] --they apply formal differentiation (Section 3.2.3.2.3) to achieve fusion; this also occurs in the garbage collection algorithm development (Section 2.4.5). In a similar manner, CIP developments (Section 2.2.4) also use fusion as a development objective.

3.2.3.1.2 Tupling

Whereas fusion merges nested function calls or loops, tupling merges parallel function calls or loops so that their independent computations may be performed collectively, and so that their common computations need not be repeated.

The research cited above for fusion also uses tupling as an objective to structure developments. In addition, the tupling tactic is used to describe transformations in the linear recurrence relation development (Section 2.4.3); tupling may save both computation time and memory when applied to functions that visit the same data structure [Pettorossi 77][4].

[4] Pettorossi consistently uses the term ·strategy· for what I am calling a ·tactic·.

3.2.3.1.3 Generalization and specialization

Generalization is a technique to solve a problem by considering a more general one. It is commonly used in mathematics and computer science, and has broad application in transformational developments.

- In the CIP group's developments, generalization (there called **embedding**) is applied frequently; its usual manifestation is the definition of a new function in terms of existing ones by the addition of further parameters or results [Bauer et al 77]. This occurs in both the developments cited in Section 2.4.2.
- Darlington implemented some simple forms of automatic generalization in his transformation system extensions (Section 3.2.1); his mechanism indiscriminately proposes generalizations of intermediate expressions to be made into new functions, querying the programmer on whether or not to pursue their development [Darlington 81b].
- Manna and Waldinger also implemented some automatic generalizations in their synthesis systems (Section 2.3.1); their mechanism tries to detect when two intermediate goals (in the form of expressions to be computed) are both instances of a more general goal, in which case their system then seeks to synthesize a program to solve that generalized goal. Similarly, Wegbreit uses the objective of matching a goal and subgoal (so as to achieve recursion) to dictate the precise generalization to make the match possible [Wegbreit 76] (akin to Darlington's use of “fold” as a pivotal step to direct formation of new functions section 3.2.1).
- Wand has a different technique for deciding when and how to generalize – it is done specifically to introduce “continuations” (data structures representing the future course of computations) [Wand 80].

Specialization is in some ways the complement of generalization. Its idea is to take advantage of the context in which some value is being computed to tailor that computation to the context, with the objective of realizing a more efficient computation of the same value.

Scherlis makes explicit use of specialization in the construction and explanation of developments (Section 2.4.4), and has formulated a set of transformation rules (akin to the fold/unfold rules of Burstall and Darlington --Section 3.2.1) whose applications may be composed to achieve specialization [Scherlis 81].

3.2.3.1.4 Filter Promotion

Filter promotion (also called **operator incorporation**) is potentially applicable to a specification or a portion of a specification in “generate and test” form. Its effect is to merge the filter testing into the generation process, and it is thus a special case of **fusion**. In some cases it may improve efficiency dramatically, by guiding the generation process and by performing early testing (and rejection where appropriate) of whole classes of partially generated items.

- A canonical example of filter promotion is the derivation of conventional sorting algorithms from the specification of sorting as “generate all permutations of the input and select an ordered one.” For an example, see Darlington's development (Section 2.4.1).
- Bird's choice of notation in an applicative language setting [Bird 84] tends to **make opportunities for promotion more discernible**, and may ease the definition and application of transformations that achieve promotion.
- Specification languages with a procedural flavor often provide convenient notations for defining generators and filters (for example, the Kestrel group's V language, Section 2.3.2, and ISI's Gist language, Section 3.2.4.2). When the generator and filter are separated by intervening statements, **propagation** is used to push the filter across those statements towards the generator.

3.2.3.1.5 Removal

Instances of forms of computation that are convenient in specification or intermediate stages in development, but can typically be replaced by lower-level more efficient code, are often used as goals for **removal**.

Removing **recursion** has been studied for some considerable time, for example, the schema-based approach of Darlington and Burstall replaced some instances of recursion with iteration [Darlington & Burstall 76]. (Note: this is **not** the same as their fold/unfold method, which remained within the recursive language level.) Many CIP developments (Section 2.2.4) have continued through to the procedural level, emphasizing recursion removal along the way.

Removing **nondeterminism** (**implicit backtracking**) can lead to major efficiency improvements. The savings can be achieved if it is possible to formulate simple tests that predict, at nondeterministic choice points, which choices lead to failure, and thus avoid the expense of going down those paths and having to backtrack later. This is done for improving programs written as goals in a space generated by sets of condition-action pairs [Sintzoff 76], for incremental construction of nondetermistically defined sets (in SETL developments -section 3.2.3.2.3, [Sharir 82]), for elimination of nondeterministic activity pruned by constraints (in the Gist language – section 3.2.4.2, [Balzer 81] and [London & Feather 82]) and as a prerequisite to other transformations (in CIP developments – section 2.4.2, [Partsch 84b]).

3.2.3.1.6 Precomputation

When some but not all of the data to be given to a program is known in advance, the program may be partially processed with that known data to give a “residual” program which can be run later on the remaining data.

This goal is often called **partial evaluation**. Ershov prefers the term **mixed computation**, and has investigated its achievement through program transformation, studying compilation as a source of examples. Using program transformation to effect the changes (as opposed to applying partial evaluator mechanisms of a fully-automatic nature) offers more scope for radical changes and improvements, however requires programmer intervention. In this use, the goal of precomputation is a transformation tactic. Similar studies are reported in [Jorring & Scherlis 86].

3.2.3.2 Transformation tactics on data

Another class of transformation tactics are those that seek to affect the maintenance and retrieval of data. Three tactics for improving efficiency of data maintenance are **memoizing** (section 3.2.3.2.1), in which dynamically computed results are saved for future retrieval; **tabulation** (section 3.2.3.2.2), in which intermediate computations are carefully ordered to lead toward the goal while avoiding recomputation and minimizing storage; and **formal differentiation** (section 3.2.3.2.3), in which results that depend on changing data are incrementally maintained as those changes take place. Finally, manipulations of **abstract data types** have also been applied as a tactic in developments (section 3.2.3.2.4).

3.2.3.2.1 Memoizing

Memoizing avoids recomputation of expressions by storing the results of evaluations the first time they are computed, and retrieving the stored values upon subsequent requests for the same computation. Function calls are usually chosen as the points for memoizing, as in Michie's “memo functions” [Michie 67]. Mostow and Cohen consider memoizing in a non-applicative context, where

stored results may be invalidated over time, and try to identify what is appropriate to memoize and when [Mostow & Cohen 85].

3.2.3.2.2 Tabulation

Tabulation is a specialized form of memoizing where the context is a goal of computing a single result or table of results. Computation of the intermediate results to reach that goal may be analyzed and reordered to reduce storage space as well as minimize unnecessary recomputation.

Bird [Bird 80] studies tabulation in a purely applicative framework and identifies three different types. The first is when the recursive calling structure ("dependency graph") exhibits uniformity, which can be exploited to optimize the storage requirements.[5] The second is used when the calling structure is not uniform, but can be embedded in a uniform structure (essentially, generalizing the function being computed). The third is the default form of tabulation that avoids recomputation of results but may be expensive in terms of storage.

3.2.3.2.3 Formal differentiation

Formal differentiation aims only to maintain computed results incrementally, as the values upon which they depend gradually change. This tactic, also known as **finite differencing**, is a generalization of **strength reduction** and **iterator inversion** [Earley 76]. In the SETL framework (Section 2.1.1), Paige, Koenig, and Sharir have studied the theory and application of this tactic quite deeply; see, for example, [Paige & Koenig 82] and [Sharir 82].

Paige's implemented system performs formal differentiation on SETL programs to efficiently compute a wide variety of set expressions. Paige uses estimates of costs of SETL operations (union, intersection, addition/deletion of single elements, set retrievals, etc.) to determine when formal differentiation will realize a speedup. While Paige appears to favor a bottom-up approach to extending the compilation process (Section 2.1.2), Sharir has studied how formal differentiation can be applied at the higher levels of specification, where it may be used to radically alter the structure of the algorithm. He has shown its use for loop fusion and for deriving deterministic algorithms for a class of nondeterministic problems (classes of nondeterministic searches over powersets), for example, transitive closure of a directed graph.

3.2.3.2.4 Transformations on abstract data types

Abstract data types provide encapsulation and abstraction in specifications and programs. Substituting appropriate concrete representations for abstract data types permits the separate local optimization of these constructs. More radical improvements become possible if aggregations of abstract data types can be formed and optimized as a whole. The encapsulations provided by abstract data types serve as useful notions around which to structure development.

Transformations for optimizing individual abstract data types and for altering their boundaries are outlined in [Wile 81], and their use is demonstrated through the development of heapsort. Many of the CIP developments manipulate abstract data types and use them in structuring (section 2.4.2). Scherlis also has accumulated a set of type transformations [Scherlis 85], which have been applied in development of a simple compiler [Jorring & Scherlis 86].

---

[5] This seems to be related to the tupling tactic, Section 3.2.3.1.2, in which uniform dependency graphs may be used to group together several previously separate computations in such a way that evaluation proceeds using only that group of values, and without the need to recompute them later.

3.2.4 Structuring the transformation catalog

Many approaches to transformation employ a large catalog of transformations, the canonical example of which is [Standish et al 76]. The focus of this section is means for **structuring** such a catalog.

3.2.4.1 Stages

The transformation process is divided into successive stages, each of which has some limited purpose. During any single stage, only a readily identifiable and/or precomputed subset of the catalogue of transformations will be potentially applicable.

Boyle (Section 2.1.3) decomposes the transformation task in a top-down fashion, into a sequence of small stages. The higher level decompositions are **between major language levels** (e.g., pure applicative LISP -> recursive FORTRAN -> FORTRAN 66). These language levels are decomposed further, until each stage is relatively trivial (e.g., renaming duplicate lambda-variables). The set of transformations applicable at each stage is small, and is applied exhaustively; essentially, each stage converts the specification into a new canonical form.

3.2.4.2 Language constructs as foci

For some specification languages, organizing the transformation catalogue around the language constructs may provide a useful index into the catalogue, provided that developments can be organized around the instances of language constructs in the specification.

Most of the Irvine program transformation catalog [Standish et al 76] is structured in this manner, with categories of transformations for dealing with assignments, gotos and labels, conditionals, etc. These categories are all based on programming-level constructs. The same structuring may also be achieved with categories based on high-level specification constructs. This is done for the ISI group's specification language Gist, which includes language constructs for expressing non determinism and constraints, arbitrary reference to past and future computation, unlimited access to global information, etc. Transformational developments of Gist specifications are generally organized around incremental removal of uses of these constructs, [London & Feather 82].

3.3 Language support

The languages for expressing specifications, programs, transformations, and developments have a major influence on the style and range of applicability of transformation research.

3.3.1 Languages for specifications and programs

The languages used to express specifications and programs divide into the following styles: wide-spectrum languages, narrow-spectrum languages, and languages extended by transformation.

3.3.1.1 Wide-spectrum languages

A w**ide-spectrum language** incorporates a variety of constructs, from high-level specification constructs down to low-level, machine-oriented ones, to permit expression of a broad range of styles of programs. Further, the intent is that transformation be done incrementally, that is, fragments of a specification may be transformed to replace a use of a higher level construct with the use of an equivalent, but lower level one. Thus it must be possible to mix constructs from different levels, at least to some extent.

This has been the goal of the CIP group (Section 2.2.4), who have developed one such language, **CIP-L**; their project is described in [CIP 84]. CIP-L supports the range from specifications using

algebraically defined data types, predicate logic, nondeterminism, etc., to classical procedural programs (variables, assignment, etc.), and even down to the control-oriented level of machine programming (labels and jumps). Their transformational developments have spanned all these levels. Definitional transformations are used to give semantic meaning to the layers of CIP-L constructs that extend from a small kernel for logic and functional programming.

Other examples are the SETL language (Section 2.1.1), which incorporates dictions ranging from the concrete level of FORTRAN up to the abstract level of set theory; and the V language (Section 2.3.2), which also has a procedural level as its lowest level, but extends from there to include logic-oriented descriptions of objects with rules to manipulate those objects, and demons and assertions operating over the database of those objects.

3.3.1.2 Narrow-spectrum languages

At the opposite extreme to wide-spectrum languages, narrow-spectrum languages pick some relatively narrow style of program or specification description and focus on finding notations and manipulations to support the expression and application of transformations within that style.

The proliferation of applicative languages are instances of narrow-spectrum languages; they admit to easy manipulation and analysis, and while they are limited in the range of programming styles that they can conveniently describe, they are capable of encompassing a range from the simple, clear structure of specifications to the intertwined structure of applicative programs that correspond to efficient procedural algorithms.

3.3.1.3 Languages extended by transformation

Sometimes transformation techniques are used to extend a language with notations tailored to the task domain. Such is the approach embodied in the Harvard PDS (Program Development System) [Cheatham et al 81], a programming support environment to aid the programmer in interactively defining the meaning of new task-specific notations. Successive refinements, mostly done by a series of transformation steps, establish these definitions, and are recorded by the system for use in replaying the refinement on a modified specification. This approach has been applied to the construction and maintenance of some sizeable systems.

Boyle's TAMPR system (Section 2.1.3) serves a similar purpose for small language extensions.

3.3.2 Language support for transformations and developments

The following are illustrations of some of the techniques used in expressing transformations and developments.

3.3.2.1 Replacement rules and pattern matching

Many transformations are expressible as **rewrite rules**, where application of one of these rules requires **matching** the left hand side of the rule, a pattern, with a fragment of the program; if the match is successful, the bindings formed in the match are substituted into the right hand side of the rule to result in a fragment to replace the matched fragment in the program. Pattern languages of varied sophistication have been used to define these rules. Pattern variables that can be matched to structures in the program grammar such as sequences or sets of statements, identifiers, arguments etc., are frequently used. Not all transformations can be expressed conveniently as replacement rules, but rather must be coded procedurally, for example, formal differentiation (Section 3.2.3.2.3).

3.3.2.2 Representing programming cliches

Programming cliches, also referred to as programming paradigms, are **standard methods of solving programming problems**. Some research encodes cliches within transformations, so that

application of such a transformation results in the installation of its cliched method of problem solving, which can then be further transformed as appropriate.

Research into representing and manipulating cliches has been the keystone of the Programmer's Apprentice project [Waters 82]. This project aims to construct an intelligent computer assistant for programmers, and as such is an instance of an alternative paradigm for programming that, like program transformation, seeks to introduce automation into the programming process. Programming cliches are represented in an Artificial Intelligence plan-like manner [Rich 81].

Smith studies the synthesis of divide-and-conquer algorithms, using **program schemes** to represent and manipulate this cliche [Smith 85].

3.3.2.3 Development languages

Development languages have already been discussed in Section 2.2 on the metaprogramming approach, where sizeable developments are represented as programs in a machine-manipulable language.

Disclaimer and Acknowledgments

This survey aims to provide a representative sampling of program transformation research, and cannot hope to be complete or up to date.

The author would like to thank his colleagues from ISI and from his earlier days at the University of Edinburgh for their interest, knowledge, and enthusiasm in program transformation and related research. Helmut Partsch provided constructive comments on an earlier draft. The NATO Advanced Research Workshop on Program Transformation and Programming Environments, Munich, September, 1983 (reported in [Pepper 83]), and meetings of IFIP WG2.1 have been particularly worthwhile sources of inspiration. Partsch and Steinbruggen's survey paper [Partsch & Steinbruggen 83], which classifies program transformation research along the dimensions of transformation systems, was of immense value.

References

[Balzer 81] Balzer, R., “Transformational Implementation: An Example,” *IEEE Transactions on Software Engineering* SE-7, (1), 1981, 3-14.

[Balzer, Cheatham & Green 83] Balzer, R., Cheatham, T.E. Jr. & Green, C., “Software technology in the 1990's: Using a new paradigm,” *Computer*, November 1983, 39-45.

[Balzer, Goldman & Wile 76] Balzer, R., Goldman, N. & Wile, D., “On the transformational implementation approach to programming,” in *Proceedings, 2nd International Conference on Software Engineering*, San Francisco, California, pp. 337-344, October 1976.

[Barstow 79] Barstow, D.R., *Knowledge-Based Program Construction*, Elsevier North Holland, 1979.

[Barstow 85] Barstow, D.R., “On convergence toward a data base of programming rules,” *ACM TOPLAS* 7, (1), January 1985,1-9.

[Barstow & Kant 76] Barstow, D.R. & Kant, E., “Observations on the interaction between coding and efficiency knowledge in the PSI program synthesis system,” in *Proceedings, Second International Conference on Software Engineering*, San Francisco, California, pp. 19-31, IEEE, 1976.

[Bates & Constable 85] Bates J.L. & Constable, R.L., “Proofs as programs,” *ACM TOPLAS* 7, (1), January 1985, 113-136.

[Bauer 76] Bauer, F .L., “Programming as an evolutionary process,” in *Proceedings, Second International Conference on Software Engineering*, San Francisco, California, pp. 223-234, IEEE, 1976.

[Bauer et al 77] Bauer, F.L., Partsch, H., Pepper, P. & Wossner, H., “Techniques for program development,” in *Infotech State of the Art Report, Software Engineering Techniques*, pp. 25-50, Infotech Information Ltd, Maidenhead, Berkshire, England, 1977.

[Bird 80] Bird, R.S., “Tabulation techniques for recursive programs,” *Computing Surveys* 12, (4), December 1980, 403-417.

[Bird 84] Bird, R.S., “The promotion and accumulation strategies in transformational programming,” *ACM TOPLAS* 6, (4), October 1984, 487-504.

[Boyle 79] Boyle, J.M., “Program adaption and program transformation,” in *Practice in Software Adaption and Maintenance: Proceedings, Workshop on Software Adaption and Maintenance*, Berlin, pp. 3-20, North-Holland, 1979.

[Boyle 84] Boyle, J.M., “Lisp to Fortran - program transformation applied,” in P. Pepper (ed.), *NATO ASI Series F: Computer and Systems Sciences. Volume 8: Program Transformation and Programming Environments. Report on a Workshop directed by F.L. Bauer and H. Remus*, pp. 291-298, Springer-Verlag, 1984.

[Broy 83] Broy, M., “Program construction by transformation: a family tree of sorting programs,” in Biermann, A.W., et al (eds.), *NATO ASI Series. Volume 95: Computer program synthesis methodologies, Proceedings, NATO Advanced Study Institute*, Bonas, Sept 28 -Oct 101981, pp. 1-49, Dordrecht: Reidel, 1983.

[Broy & Bruckner 80] Broy, M. & Krieg-Bruckner, B., “Derivation of invariant assertions during program development by transformation,” *ACM TOPLAS* 2, (3), July 1980, 321-337.

[Broy & Pepper 82] Broy, M. & Pepper, P., “Combining algebraic and algorithmic reasoning: an approach to the Schorr-Waite algorithm,” *ACM TOPLAS* 4, (3), July 1982, 362-381.

[Burstall & Darlington 77] Burstall, R.M. & Darlington, J., “A transformation system for developing recursive programs,” *JACM* 24, (1), January 1977, 44-67.

[Burstall & Feather 77] Burstall, R.M. & Feather, M.S., “Program development by transformations: an overview,” in *Proceedings, Lea Fondements de la Programmation*, Toulouse, France, pp. 45-55, 1977.

[Cheatham et al 81] Cheatham, T.E.Jr., Holloway, G.H. & Townley, J.A., “Program refinement by transformation,” in *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, California, pp. 430-437, March 1981.

[CIP 84] CIP Language Group, *Lecture Notes in Computer Science*. Volume I: The Munich project CIP, Springer-Verlag, 1984.

[Darlington 75] Darlington, J., “Applications of program transformation to program synthesis,” in *Proceedings of International Symposium on Proving and Improving Programs*, Arc-et-Senans, France, pp. 133-144, 1975.

[Darlington 78] Darlington, J., “A synthesis of several sorting algorithms,” *Acta Informatica* 11, (1), December 1978, 1-30.

[Darlington 81a] Darlington, J., “The structured description of algorithm derivations,” in deBakker, J.W. & van Vliet, H. (eds.), *Algorithmic Languages*, pp. 221-250, Elsevier North-Holland, New York, 1981.

[Darlington 81b] Darlington, J., “An experimental program transformation and synthesis system,” *Artificial Intelligence* 16, (1), March 1981, 1-46.

[Darlington & Burstall 76] Darlington, J. & Burstall, R.M., "A system which automatically improves programs," *Acta Informatica* 6, (1), March 1976, 41-60.

[Darlington & Feather 80] Darlington, J. & Feather, M.S., *A transformational approach to program modification*, Department of Computing and Control, Imperial College, London, Technical Report 80/3, 1980.

[Dershowitz 85] Dershowitz, N., "Synthetic programming," *Artificial Intelligence* 25, 1985, 323-373.

[Dewar et al 79] Dewar, R.B.K., Grand, A., Liu, S. & Schwartz, J.T., "Programming by refinement, as exemplified by the SETL representation sublanguage," *ACM TOPLAS* 1, (1), 1979, 27-49.

[Dewar et al 81] Dewar, R.B.K., Schonberg, E. & Schwartz, J.T., *Higher level programming: introduction to the use of the set-theoretic programming language SETL*, Courant Institute of Mathematical Sciences, New York University, New York, Technical Report, 1981.

[Dewar, Sharir & Weixelbaum 82] Dewar, R.B.K., Sharir, M. & Weixelbaum, E., "Transformational derivation of a garbage collection algorithm," *ACM TOPLAS* 4, (4), October 1982, 650-667.

[Earley 76] Earley, J., "High level iterators and a method for automatically designing data structure representation," *Computer Languages* 1, (4), 1976,321-342.

[Feather 82a] Feather, M.S., "A system for assisting program transformation," *ACM TOPLAS* 4, (1), January 1982, 1-20.

[Feather 82b] Feather, M.S., "Program specification applied to a text-formatter," *IEEE Transactions on Software Engineering* SE-8, (5), September 1982, 490-498.

[Fickas 82] Fickas, S.F., *Automating the transformational development of software*, Ph.D. thesis, University of California, Irvine, 1982.

[Freudenberger et al 83] Freudenberger, S.M., Schwartz, J.T. & Sharir, M., "Experience with the SETL optimizer," *ACM TOPLAS* 5, (1), January 1983, 26-45.

[Gerhart 75] Gerhart, S.1., "Correctness preserving program transformations," in *Proceedings, 2nd ACM POPL Symposium*, Palo Alto, California, pp. 54-66, 1975.

[Gordon et al 78] Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. & Wadsworth, C., "A metalanguage for interactive proof in LCF," in *Proceedings, 5th ACM POPL Symposium*, Tucson, Arizona, pp. 119-130, 1978.

[Green 77] Green, C.C., "A summary of the PSI program synthesis system," in *Proceedings of the Fifth International Conference on Artificial Intelligence*, Cambridge, Massachusetts, pp. 380-381, August 1977.

[Green & Barstow 78] Green, C. & Barstow, D., "On program synthesis knowledge," *Artificial Intelligence* 10, 1978, 241-279.

[Green et al 81] Green, C., Phillips, J., Westfold, S., Pressburger, T., Kedzierski, B., Angebranndt, S., Mont-Reynaud, B. & Tappel, S., *Research on knowledge-based programming and algorithm design*, Kestrel Institute, Palo Alto, California, Technical Report KES.U.81.2, August 1981.

[Horsch et al 85] Horsch, A., Moller, B., Partsch, H., Paukner, O. & Pepper, P., *The Munich project CIP Volume II: The program transformation system CIP-S, Part I: formal specification (Tentative version)*, Institut fur Informatik, Technische Universitat Munchen, Technical Report TUM-18509, 1985.

[Jorring & Scherlis 86] Jorring, U. & Scherlis, W.L., “Compilers and Staging Transformations,” in to appear in: *Proceedings, ACM Symposium on the Principles of Programming Languages* 1986, pp. 1, 1986.

[Kant 83] Kant, E., “On the efficient synthesis of efficient programs,” *Artificial Intelligence* 20, 1983, 253-305.

[Kant & Barstow 81] Kant, E. & Barstow, D.R., “The refinement paradigm: the interaction of coding and efficiency knowledge in program synthesis,” *IEEE TSE* 7, 1981, 458-471.

[London & Feather 82] London, P.E. & Feather, M.S., “Implementing specification freedoms,” S*cience of Computer Programming*, (2), 1982, 91-131.

[Manna & Waldinger 79] Manna, Z. & Waldinger, R., “Synthesis: dreams => programs,” *IEEE Transactions on Software Engineering* SE-5, (4), 1979,294-328.

[Manna & Waldinger 80] tv1anna, Z. & Waldinger, R., “A deductive approach to program synthesis,” ACM TOPLAS 2, (1), 1980, 90-121.

[Manna & Waldinger 81] Manna, Z. & Waldinger, R., “Deductive synthesis of the unification algorithm,” *Science of Computer Programming* 1, 1981, 5-48.

[Michie 67] Michie, D., *Memo functions: A language feature with rote learning properties, Department of Machine Intelligence and Perception*, University of Edinburgh, Scotland, Technical Report DMIP Memo MIP-R-29, 1967.

[Mostow 81] Mostow, D.J., *Mechanical transformation of task heuristics into operational procedures,* Ph.D. thesis, Computer Science Department, CarnegieMellon University; CMU-CS-81-113, 1981.

[Mostow 83] Mostow, D.J., “A problem-solver for making advice operational,” in *Proceedings, Srd National Conference on Artificial Intelligence*, Washington D.C., August 1983.

[Mostow & Cohen 85] Mostow, D.J. & Cohen, D., “Automating program speedup by deciding what to cache,” in Joshi, A. (ed.), *Proceedings, 9th International Joint Conference on Artificial Intelligence*, Los Angeles, pp. 165-172, August 1985.

[Paige 83] Paige, R., “Transformational programming - applications to algorithms and systems,” in *Proceedings, 10th ACM POPL Symposium*, Austin, Texas, pp. 73-87, 1983.

[Paige 84] Paige, R., “Supercompilers -extended abstract,” in P. Pepper (ed.), *NATO ASI Series F: Computer and Systems Sciences. Volume 8: Program Transformation and Programming Environments. Report on a Workshop directed by F.L. Bauer and H. Remus*, pp. 331-340, Springer-Verlag, 1984.

[Paige & Koenig 82] Paige, R. & Koenig, S., “Finite differencing of computable expressions,” *ACM TOPLAS* 4, (3), July 1982, 402-454.

[Partsch 83a] Partsch, H., *A transformational approach to parsing and recognition*, Institut fur Informatik, Technische Universitat Munchen, Technical Report TUM 18314, 1983.

[Partsch 83b] Partsch, H., “An exercise in the transformational derivation of an efficient program by joint development of control and data structure,” *Science of Computer Programming*, (3), 1983, 1-35.

[Partsch 84a] Partsch, H., “The CIP transformation system,” in P. Pepper (ed.), *NATO ASI Series F: Computer and Systems Sciences. Volume 8: Program Transformation and Programming Environments. Report on a Workshop directed by F.L. Bauer and H. Remus*, pp. 305-322, Springer-Verlag, 1984.

[Partsch 84b] Partsch, H., “Structuring transformational developments: a case study based on Earley's recognizer,” *Science of Computer Programming*, (4), 1984, 17-44.

[Partsch & Steinbruggen 83] Partsch, H. & Steinbruggen, R., “Program transformation systems,” *Computing Surveys* 15, (3), 1983, 199-236.

[Pepper 83] Edited by P. Pepper, *NATO ASI Series F: Computer and Systems Sciences. Volume 8: Program transformation and programming environments, Springer Verlag, 1983. Report on a workshop directed by F.L. Bauer and H. Remus*.

[Pepper & Partsch 80] Pepper, P. & Partsch, H., *On the feedback between specifications and implementations: an example,* Institut fur Informatik, Technische Universitat Munchen, Technical Report TUM-I8011, 1980.

[Pettorossi 77] Pettorossi, A., “Transformation of programs and use of 'tupling strategy',” in *Proceedings, Informatica 77,* Bled, Yugoslavia, pp. 1-6, 1977.

[Pettorossi & Burstall 82] Pettorossi, A. & Burstall, R.M., “Deriving very efficient algorithms for evaluating recurrence relations using the program transformation technique,” *Acta Informatica* 18, 1982, 181-206.

[Reif & Scherlis 83] Reif, J.H. & Scherlis, W.L., “Deriving efficient graph algorithms,” in Clarke & Kozen (eds.), Lecture notes in computer science, No. 164. : *Proceedings, Logics of Programs Workshop,* Carnegie Mellon University, Pittsburgh, pp. 421-441, Springer-Verlag, 1983.

[Rich 81] Rich, C., “A formal representation for plans in the programmer's apprentice,” in *Proceedings, 7th International Joint Conference on Artificial Intelligence,* pp. ???, 1981.

[Scherlis 80] Scherlis, W.L., *Expression procedures and program derivation*, Ph.D. thesis, Stanford University, 1980. Stanford AI Lab Memo AIM-341, Dept of Computer Science Report STAN-CS-80-818

[Scherlis 81] Scherlis, W.L., “Program Improvement by Internal Specialization,” in *Proceedings, Eighth ACM Symposium on the Principles of Programming Languages,* Williamsburg, VA, pp. 41-49, January 1981.

[Scherlis 85] Scherlis, W.L., *Adapting abstract data types*, Carnegie-Mellon University, Technical Report , 1985.

[Scherlis & Scott 83] Scherlis, W.L. & Scott, D.S., “First steps towards inferential programming,” in *Proceedings, IFIP 83 Congress*, North Holland, 1983.

[Schmitz 82] Schmitz, L., “An exercise in program synthesis: algorithms for computing the transitive closure of a relation,” *Science of Computer Programming* 1, (3), 1982, 235-254.

[Sharir 81] Sharir, M., “A strange sorting technique inspired by program transformation,” *Computational Mathematics Applications* 7, (4), 1981,293-298.

[Sharir 82] Sharir, M., “Some observations concerning formal differentiation of set theoretic expressions,” *ACM TOPLAS* 4, (2), April 1982, 196-225.

[Sintzoff 76] Sintzoff, M., “Eliminating blind alleys from backtrack programs,” in *Automata Languages and Programming, Third International Colloqium*, University of Edinburgh, pp. 531-557, Edinburgh University Press, July 1976.

[Sintzoff 80] Sintzoff, M., “Suggestions for composing and specifying program design decisions,” in *4th International Symposium on Programming*, Paris, April 1980.

[Smith 85] Smith, D.R., “The design of divide and conquer algorithms,” *Science of Computer Programming* 5, (1), 1985, 37-58.

[Smith et al 85] Smith, D.R., Kotik, G.B. & Westfold, S.J., “Research on knowledgebased software environments at Kestrel Institute,” *IEEE TSE* SE-11, (11), 1985, 1278-1295.

[Standish et al 76] Standish, T.A., Harriman, D.C., Kibler, D.F. & Neighbors, J.M., *The Irvine program transformation catalogue*, University of California, Irvine, Technical Report, 1976.

[Steinbruggen 82] Steinbruggen, R., *Program development using transformational expressions*, Institut fur Informatik, Technische Universitat Munchen, Technical Report TUM-18206, 1982.

[Swartout & Balzer 82] Swartout, W. & Balzer, R., "On the inevitable intertwining of specification and implementation," *CACM* 25, (7), 1982, 438-440.

[Waldinger 77] Waldinger, R.J., "Achieving several goals simultaneously," in Elcock & Michie (ed.), *Machine Intelligence 8: Machine Representations of Knowledge*, pp. 94-136, Ellis Horwood Ltd, 1977.

[Wand 80] Wand, M., "Continuation-based program transformation strategies," *JACM* 27, (1), 1980, 164-180.

[Waters 82] Waters, R.C., "The programmer's apprentice: knowledge based program editing," *IEEE Transactions on Software Engineering* SE-8, (1), January 1982, 1-12.

[Wegbreit 76] Wegbreit, B., "Goal-directed program transformation," *IEEE Transactions on Software Engineering* SE-2, (2), 1976, 69-80.

[Wile 81] Wile, D.S., "Type transformations," *IEEE TSE* SE-7, (1), January 1981, 32-39.

[Wile 82] Wile, D.S., *POPART: Producer of Parsers and Related Tools, System Builders' Manual*, ISI, 4676 Admiralty Way, Marina del Rey, CA 90292, Technical Report RR-82-21, 1982.

[Wile 83] Wile, D.S., "Program developments: formal explanations of implementations," *CACM* 26, (11), November 1983, 902-911.